THE EFFECTIVE ENGINEER

How to Leverage Your Efforts in Software Engineering to Make a Disproportionate and Meaningful Impact

Edmond Lau

Reviews for the The Effective Engineer

"I wish I had this manual to give my engineers when I was a VP of Engineering at Twitter. This summarizes and presents everything I used to tell my team."

— Raffi Krikorian, Former VP of Platform Engineering at Twitter

"…one of the few titles written from an engineer's perspective with enough specific information to help you and your team work better. In particular I enjoyed the section on metrics and how choosing the right ones will give teams the clear direction they need. It's a worthwhile read."

— Mike Curtis, VP of Engineering at Airbnb

"I hope more people embrace Edmond's philosophy and techniques to make their companies and careers more successful."

— Bret Taylor, CEO of Quip & Former CTO of Facebook

"I've continuously relayed Edmond's advice to my teammates throughout the years, but I'm glad that I can do so in book form, without the lossiness of my transmission."

— Tracy Chou, Tech Lead at Pinterest

"I'd always thought that good engineers were born of long, hard experience, so I didn't think a book could teach me how to be more effective. But in fact, Edmond managed to distill his decade of engineering experience into crystal-clear best practices."

— Daniel Peng, Senior Staff Engineer at Google

"Much of being a great engineer, and building a great engineering team, is easier said than done, but *The Effective Engineer* is a comprehensive tour of our industry's collective wisdom written with clarity."

— Jack Heart, Engineering Manager at Asana

4

Invest in Iteration Speed

On any given day, our team at Quora might release new versions of the web product to users 40 to 50 times. [1] Using a practice called *continuous deployment*, we automatically shipped any new code we committed to production servers. On average, it took only seven minutes for each change to be vetted by thousands of tests and receive the green light to roll out to millions of users. This happened throughout every day, all without any human intervention. In contrast, most other software companies shipped new releases only weekly, monthly, or quarterly, and the mechanics of each release might take hours or even days.

For those who haven't used it before, continuous deployment may seem like a scary or unviable process. How did we manage to deploy software more frequently—orders of magnitude more frequently, in fact—than other teams, without sacrificing quality and reliability? Why would we even want to release software that often? Why not hire or contract a quality assurance team to sanity check each release? To be honest, when I first joined Quora back in August 2010, I had similar concerns. New engineers add themselves to the team page as one of their first tasks, and the notion that the code I wrote on my first day would so easily go into production was exhilarating—and frightening.

But now, after having used the process for three years, it's clear to me that continuous deployment played an instrumental role in helping our team grow the product. We increased new user registrations and user engagement metrics by over 3x during my last year at Quora. Continuous deployment, along with other investments in iteration speed, contributed in large part to that growth. [2]

A number of high-leverage investments in our infrastructure made this rapid release cycle possible. We built tools to automatically version and package our code. We developed a testing framework that parallelized thousands of unit and integration tests across a tier of worker machines. If all the tests passed, our release scripts tested the new build on web servers, called *canaries*, to further validate that everything behaved as expected, and then rolled out the software to production tiers. We invested in comprehensive dashboards and alerts that monitored our product's health, and we made tools to easily roll back changes in the event that some bad code had fallen through the cracks. Those investments eliminated the manual overhead associated with each deployment and gave us high confidence that each deployment was just business as usual.

Why is continuous deployment such a powerful tool? Fundamentally, it allows engineers to make and deploy small, incremental changes rather than the larger, batched changes typical at other companies. That shift in approach eliminates a significant amount of overhead associated with traditional release processes, making it easier to reason about changes and enabling engineers to iterate much more quickly.

If someone finds a bug, for instance, continuous deployment makes it possible to implement a fix, deploy it to production, and verify that it works—all in one sitting. With more traditional workflows, those three phases might be split over multiple days or weeks; the engineer has to make the fix, wait days for it to be packaged up with other bigger changes in the week's release, and then validate that fix along with a slew of other orthogonal changes. Much more context switching and mental overhead are required.

Or suppose you need to migrate an in-production database table from one schema to another. The standard process to change a live schema is: 1) create the new schema, 2) deploy code to write to both the old and new schemas, 3) copy existing data over from the old schema to the new schema, 4) deploy code

to start reading from the new schema, and 5) remove the code that writes to the old schema. While each individual change might be straightforward, the changes need to happen sequentially over 4–5 releases. This can be quite laborious if each release takes a week. With continuous deployment, an engineer could perform the migration by deploying 4–5 times within a few hours and not have to think about it again in subsequent weeks.

Because changes come in smaller batches, it's also easier to debug problems when they're identified. When a bug surfaces or when a performance or business metric drops as a result of a release, teams with a weekly release cycle often have to dig through hundreds of changes from the past week in an attempt to figure out what went wrong. With continuous deployment, on the other hand, it generally is a straightforward task to isolate the culprit in the handful of code changes deployed in the past few hours.

Just because changes are deployed incrementally, however, doesn't mean that larger features aren't possible or that users see half-finished features. A large feature gets gated behind a configuration flag, which is disabled until the feature is ready. The same configuration flag often allows teams to selectively enable a feature for internal team members, beta users, or some fraction of production traffic until the feature is ready. In practice, this also means that changes get merged incrementally into the master code repository. Teams then avoid the intense coordination and "merge hell" that often accompanies longer release cycles as they scramble to integrate large chunks of new code and get them to work together correctly. [3]

Focusing on small, incremental changes also opens the door to new development techniques that aren't possible in traditional release workflows. Suppose that in a product discussion, we're debating whether we should keep a certain feature. Rather than letting opinions and politics dictate the feature's importance or waiting for the next release cycle to start gathering usage data, we could simply log the interaction we care about, deploy it, and start seeing the initial data trickle in within minutes. Or suppose we see a performance regression on one of our web pages. Rather than scanning through the code looking for regressions, we can spend a few minutes deploying a change to enable logging so that we can get a live breakdown of where time is being spent.

Our team at Quora wasn't alone in our strong emphasis on the importance of iterating quickly. Engineering teams at Etsy, [4] IMVU, [5] Wealthfront, [6] and GitHub, [7] as well as other companies, [8] have also incorporated continuous deployment (or a variant called *continuous delivery,* where engineers selectively determine which versions to deploy) into their workflows.

Effective engineers invest heavily in iteration speed. In this chapter, we'll find out why these investments are so high-leverage and how we can optimize for iteration speed. First, we'll discuss the benefits of iterating quickly: we can build more and learn faster. We'll then show why it's critical to invest in time-saving tools and how to increase both their adoption and your leeway. Since much of our engineering time is spent debugging and testing, we'll walk through the benefits of shortening our debugging and validation loops. Most of our core tools remain the same throughout our career, so we'll also review habits for mastering our programming environments. And lastly, because programming is but one element in the software development process, we'll look at why it's also important to identify the non-engineering bottlenecks in your work.

Move Fast to Learn Fast

In the hallways of Facebook's Menlo Park headquarters, posters proclaim in red caps: "MOVE FAST AND BREAK THINGS." This mantra enabled the social network to grow exponentially, acquiring over 1 billion users in just 8 years. [9] New employees are indoctrinated into the culture of moving fast during Bootcamp, Facebook's 6-week onboarding program. [10] Many new employees, including those who've never before used PHP, the website's primary programming language, ship code to production in their first few days. Facebook's culture emphasizes iterating quickly and focusing on impact rather than being conservative and minimizing mistakes. The company might not use continuous deployment in production, but it has managed to effectively scale its workflow so that over a thousand engineers are able to deploy code to facebook.com twice a day. [11] That's an impressive feat.

Facebook's growth illustrates why investing in iteration speed is such a high-leverage decision. The faster that you can iterate, the more that you can learn about what works and what doesn't work. You can build more things and try out more ideas. Not every change will produce positive value and growth, of course. One of Facebook's early advertising products, Beacon, automatically broadcasted a user's activity on external websites onto Facebook. The product caused an uproar and had to be shut down. [12] But with each iteration, you get a better sense of which changes will point you in the right direction, making your future efforts much more effective.

Facebook CEO Mark Zuckerberg captured the importance of moving fast in his letter that accompanied the company's initial public offering. "Moving fast enables us to build more things and learn faster," he wrote. "However, as most companies grow, they slow down too much because they're more afraid of making mistakes than they are of losing opportunities by moving too slowly … [I]f you never break anything, you're probably not moving fast enough." [13] A strong focus on maintaining a high iteration speed is a key ingredient for how Facebook got to where it is today.

Moving fast isn't just restricted to consumer web software, where users tend to more tolerant of downtime. And in actuality, the worst outage Facebook ever faced over a four-year period lasted only 2.5 hours—much shorter than outages experienced by larger, slower-moving companies. [14] Moving fast doesn't necessarily mean moving recklessly.

Consider Wealthfront, a financial advisory service whose offices are located in Palo Alto, CA. Wealthfront is a technology company whose mission is to provide access to the financial advice offered by major financial institutions and private wealth managers, at a low cost. They do this by replacing human advisors with software-based ones. As of June 2014, the company manages over a billion dollars in customer assets. [15] Any code breakage would be very costly—but despite this, Wealthfront has invested in continuous deployment and uses the system to ship new code to production over 30 times per day. [16] They're able to iterate quickly despite operating in a financial space that's heavily regulated by the Securities and Exchange Commission and other authorities. Pascal-Louis Perez, Wealthfront's former CTO, explained that continuous de-

ployment's "primary advantage is risk reduction," as it lets the team focus on small batches of changes and "quickly pinpoint problems when they occur." [17]

Continuous deployment is but one of many powerful tools at your disposal for increasing iteration speed. Other options include investing in time-saving tools, improving your debugging loops, mastering your programming workflows, and, more generally, removing any bottlenecks that you identify. We'll spend the rest of the chapter discussing actionable steps for these strategies. All of these investments accomplish the same goal as continuous deployment: they help you move fast and learn quickly about what works and what doesn't. And remember: because learning compounds, the sooner you accelerate your iteration speed, the faster your learning rate will be.

Invest in Time-Saving Tools

When I ask engineering leaders which investments yield the highest returns, "tools" is the most common answer. Bobby Johnson, a former Facebook Director of Infrastructure Engineering, told me, "I've found that almost all successful people write a lot of tools … [A] very good indicator of future success [was] if the first thing someone did on a problem was to write a tool." [18] Similarly, Raffi Krikorian, former VP of Platform Engineering at Twitter, shared with me that he'd constantly remind his team, "If you have to do something manually more than twice, then write a tool for the third time." [19] There are only a finite number of work hours in the day, so increasing your effort as a way to increase your impact fails to scale. Tools are the multipliers that allow you to scale your impact beyond the confines of the work day.

Consider two engineers, Mark and Sarah, working on two separate projects. Mark dives head first into his project and spends his next two months building and launching a number of features. Sarah, on the other hand, notices that her workflow isn't as fast it could be. She spends her first two weeks fixing her workflow—setting up incremental compilation of her code, configuring her web server to automatically reload newly compiled code, and writing a few automation scripts to make it easier to set up a test user's state on her development server. These improvements speed up her development cycles by 33%.

Mark was able to get more done initially, but after two months, Sarah catches up—and her remaining six weeks' worth of feature work is as productive as Mark's eight weeks'. Moreover, Sarah continues to move 33% faster than Mark even after those first two months, producing significantly more output going forward.

The example is somewhat simplified. In reality, Sarah wouldn't actually front-load all that time into creating tools. Instead, she would iteratively identify her biggest bottlenecks and figure out what types of tools would let her iterate faster. But the principle still holds: time-saving tools pay off large dividends.

Two additional effects make Sarah's approach even more compelling. First, faster tools get used more often. If the only option for travel from San Francisco to New York was a week-long train ride, we wouldn't make the trip very often; but since the advent of passenger airlines in the 1950s, people can now make the trip multiple times per year. Similarly, when a tool halves the time it takes to complete a 20-minute activity that we perform 3 times a day, we save much more than 30 minutes per day—because we tend to use it more often. Second, faster tools can enable new development workflows that previously weren't possible. Together, these effects mean that 33% actually might be an *underestimate* of Sarah's speed advantage.

We've already seen this phenomenon illustrated by continuous deployment. A team with a traditional weekly software release process takes many hours to cut a new release, deploy the new version to a staging environment, have a quality assurance team test it, fix any blocking issues, and launch it to production. How much time would streamlining that release process save? Some might say a few hours each week, at most. But, as we saw with continuous deployment, getting that release time down to a few minutes means that the team can actually deploy software updates much more frequently, perhaps at a rate of 40–50 times per day. Moreover, the team can interactively investigate issues in production—posing a question and deploying a change to answer it—an otherwise difficult task. Thus, the total time saved greatly exceeds a few hours per week.

Or consider compilation speed. When I first started working at Google back in 2006, compiling C++ code for the Google Web Server and its depen-

dencies could take upwards of 20 minutes or more, even with distributed compilation. [20] When code takes that long to compile, engineers make a conscious decision not to compile very often—usually no more than a few times a day. They batch together large chunks of code for the compiler and try to fix multiple errors per development cycle. Since 2006, Google has made significant inroads into reducing compilation times for large programs, including some open source software that shorten compilation phases by 3–5x. [21]

When compile times drop from 20 minutes to, say, 2 minutes, engineering workflows change drastically. This means even an hour or two of time savings per day is a big underestimate. Engineers spend less time visually inspecting code for mistakes and errors and rely more heavily on the compiler to check it. Faster compile times also facilitate new workflows around iterative development, as it's simpler to iteratively reason about, write, and test smaller chunks of code. When compile times drop to seconds, incremental compilation—where saving a file automatically triggers a background task to start recompiling code—allows engineers to see compiler warnings and errors as they edit files, and makes programming significantly more interactive than before. And faster compile times mean that engineers will compile fifty or even hundreds of times per day, instead of ten or twenty. Productivity skyrockets.

Switching to languages with interactive programming environments can have a similar effect. In Java, testing out a small expression or function entails a batch workflow of writing, compiling, and running an entire program. One advantage that languages like Scala or Clojure, two languages that run on the Java Virtual Machine, have over Java itself is their ability to evaluate expressions quickly and interactively within a read-eval-print loop, or REPL. This doesn't save time just because the read-eval-print loop is faster than the edit-compile-run-debug loop; it also saves time because you end up interactively evaluating and testing many more small expressions or functions that you wouldn't have done before.

There are plenty of other examples of tools that compound their time savings by leading to new workflows. Hot code reloads, where a server or application can automatically swap in new versions of the code without doing a full restart, encourages a workflow with more incremental changes. Continuous in-

tegration, where every commit fires off a process to rebuild the codebase and run the entire test suite, makes it easy to pinpoint which change broke the code so that you don't have to waste time searching for it.

The time-saving properties of tools also scale with team adoption. A tool that saves you one hour per day saves 10 times as much when you get 10 people on your team to use it. That's why companies like Google, Facebook, Dropbox, and Cloudera have entire teams devoted to improving internal development tools; reducing the build time by one minute, when multiplied over a team of 1,000 engineers who build code a dozen times a day, translates to nearly one person-year in engineering time saved every week! Therefore, it's not sufficient to find or build a time-saving tool. To maximize its benefits, you also need to increase its adoption across your team. The best way to accomplish that is by proving that the tool actually saves time.

When I was working on the Search Quality team at Google, most people who wanted to prototype new user interfaces for Google's search result pages would write them in C++. C++ was a great language choice for the high performance needed in production, but its slow compile cycles and verbosity made it a poor vehicle for prototyping new features and testing out new user interactions.

And so, during my 20% time, I built a framework in Python that let engineers prototype new search features. Once my immediate teammates and I started churning out prototypes of feature after feature and demoing them at meetings, it didn't take very long for others to realize that they could also be much more productive building on top of our framework, even if it meant porting over their existing work.

Sometimes, the time-saving tool that you built might be objectively superior to the existing one, but the switching costs discourage other engineers from actually changing their workflow and learning your tool. In these situations, it's worth investing the additional effort to lower the switching cost and to find a smoother way to integrate the tool into existing workflows. Perhaps you can enable other engineers to switch to the new behavior with only a small configuration change.

When we were building our online video player at Ooyala, for example, everyone on the team used an Eclipse plugin to compile their ActionScript code, the language used for Flash applications. Unfortunately, the plugin was unreliable and sometimes failed to recompile a change. Unless you carefully watched what was being compiled, you wouldn't discover that your changes were missing until you actually interacted with the video player. This led to frequent confusion and slower development. I ended up creating a new command-line based build system that would produce reliable builds. Initially, because it required changing their build workflow off from Eclipse, only a few team members adopted my system. And so, to increase adoption, I spent some additional time and hooked the build process into Eclipse. That reduced the switching costs sufficiently to convince others on the team to change systems.

One side benefit of proving to people that your tool saves time is that it also earns you leeway with your manager and your peers to explore more ideas in the future. It can be difficult to convince others that an idea that you believe in is actually worth doing. Did the new Erlang deployment system that Joe rewrote in one week for fun actually produce any business value? Or is it just an unmaintainable liability? Compared with other projects, time-saving tools provide measurable benefits—so you can use data to objectively prove that your time investment garnered a positive return (or, conversely, to prove to yourself that your investment wasn't worth it). If your team spends 3 hours a week responding to server crashes, for example, and you spend 12 hours building a tool to automatically restart crashed servers, it's clear that your investment will break even after a month and pay dividends going forward.

At work, we can easily fall into an endless cycle of hitting the next deadline: getting the next thing done, shipping the next new feature, clearing the next bug in our backlog, and responding to the next issue in the never-ending stream of customer requests. We might have ideas for tools we could build to make our lives a bit easier, but the long-term value of those tools is hard to quantify. On the other hand, the short-term costs of a slipped deadline or a product manager breathing down our necks and asking when something will get done are fairly concrete.

So start small. Find an area where a tool could save time, build it, and demonstrate its value. You'll earn leeway to explore more ambitious avenues, and you'll find the tools you build empowering you to be more effective on future tasks. Don't let the pressure to constantly ship new features cause the important but non-urgent task of building time-saving tools to fall by the wayside.

Shorten Your Debugging and Validation Loops

It's wishful thinking to believe that all the code we write will be bug-free and work the first time. In actuality, much of our engineering time is spent either debugging issues or validating that what we're building behaves as expected. The sooner we internalize this reality, the sooner we will start to consciously invest in our iteration speed in debugging and validation loops.

Creating the right workflows in this area can be just as important as investing in time-saving tools. Many of us are familiar with the concept of a minimal, reproducible test case. This refers to the simplest possible test case that exercises a bug or demonstrates a problem. A minimal, reproducible test case removes all unnecessary distractions so that more time and energy can be spent on the core issue, and it creates a tight feedback loop so that we can iterate quickly. Isolating that test case might involve removing every unnecessary line of code from a short program or unit test, or identifying the shortest sequence of steps a user must take to reproduce an issue. Few of us, however, extend this mentality more broadly and create minimal workflows while we're iterating on a bug or a feature.

As engineers, we can shortcut around normal system behaviors and user interactions when we're testing our products. With some effort, we can programmatically build much simpler custom workflows. Suppose, for example, you're working on a social networking application for iOS, and you find a bug in the flow for sending an invite to a friend. You could navigate through the same three interactions that every normal user goes through: switching to the friends tab, choosing someone from your contacts, and then crafting an invite message. Or, you could create a much shorter workflow by spending a few min-

utes wiring up the application so that you're dropped into the buggy part of the invitation flow every time the application launches.

Or suppose you're working on an analytics web application where you need to iterate on an advanced report that is multiple clicks away from the home screen. Perhaps you also need to configure certain filters and customize the date range to pull the report you're testing. Rather than going through the normal user flow, you can shorten your workflow by adding the ability to specify the configuration through URL parameters so that you immediately jump into the relevant report. Or, you can even build a test harness that specifically loads the reporting widget you care about.

As a third example, perhaps you're building an A/B test for a web product that shows a random feature variant to users, depending on their browser cookie. To test the variants, you might hard code the conditional statement that chooses between the different variants, and keep changing what gets hard coded to switch between variants. Depending on the language you're using, this might require recompiling your code each time. Or, you can shorten your workflow by building an internal tool that lets you set your cookie to a value that can reliably trigger a certain variant during testing.

These optimizations for shortening a debugging loop seem obvious, now that we've spelled them out. But these examples are based on real scenarios that engineers at top tech companies have faced—and in some cases, they spent months using the slower workflow before realizing that they could shorten it with a little time investment. When they finally made the change and were able to iterate much more quickly, they scratched their heads, wondering why they didn't think to do it earlier.

When you're fully engaged with a bug you're testing or a new feature you're building, the last thing you want to do is to add more work. When you're already using a workflow that works, albeit with a few extra steps, it's easy to get complacent and not expend the mental cycles on devising a shorter one. Don't fall into this trap! The extra investment in setting up a minimal debugging workflow can help you fix an annoying bug sooner and with less headache.

Effective engineers know that debugging is a large part of software development. Almost instinctively, they know when to make upfront investments to

shorten their debugging loops rather than pay a tax on their time for every iteration. That instinct comes from being mindful of what steps they're taking to reproduce issues and reflecting on which steps they might be able to short circuit. "Effective engineers have an obsessive ability to create tight feedback loops for what they're testing," Mike Krieger, co-founder and CTO of the popular photo-sharing application Instagram, told me during an interview. "They're the people who, if they're dealing with a bug in the photo posting flow on an iOS app … have the instinct to spend the 20 minutes to wire things up so that they can press a button and get to the exact state they want in the flow every time."

The next time you find yourself repeatedly going through the same motions when you're fixing a bug or iterating on a feature, pause. Take a moment to think through whether you might be able to tighten that testing loop. It could save you time in the long run.

Master Your Programming Environment

Regardless of the types of software we build throughout our careers, many of the basic tools that we need to use on a daily basis remain the same. We spend countless hours working in text editors, integrated development environments (IDEs), web browsers, and mobile devices. We use version control and the command line. Moreover, certain basic skills are required for the craft of programming, including code navigation, code search, documentation lookup, code formatting, and many others. Given how much time we spend in our programming environments, the more efficient we can become, the more effective we will be as engineers.

I once worked with an engineer at Google who moused through the folder hierarchy of Mac's Finder every time he wanted to navigate to the code in another file. Say it took 12 seconds to find the file, and say he switched files 60 times per day. That's 12 minutes he spent navigating between files every day. If he had learned some text editor keyboard shortcuts that let him navigate to a file in 2 seconds instead of 12, then over the course of one day, he would have saved 10 minutes. That translates to 40 hours, or an entire work week, each year.

There are numerous other examples of simple, common tasks that can take a wide range of times for different people to complete. These include:

- Tracking changes in version control
- Compiling or building code
- Running a unit test or program
- Reloading a web page on a development server with new changes
- Testing out the behavior of an expression
- Looking up the documentation for a certain function
- Jumping to a function definition
- Reformatting code or data in text editor
- Finding the callers of a function
- Rearranging desktop windows
- Navigating to a specific place within a file

Fine-tuning the efficiency of simple actions and saving a second here or there may not seem worth it at first glance. It requires an upfront investment, and you'll very likely be slower the first few times you try out a new and unfamiliar workflow. But consider that you'll repeat those actions at least tens of thousands of times during your career: minor improvements easily compound over time. Not looking at the keyboard when you first learned to touch type might have slowed you down initially, but the massive, long-term productivity gains made the investment worthwhile. Similarly, no one masters these other skills overnight. Mastery is a process, not an event, and as you get more comfortable, the time savings will start to build. The key is to be mindful of which of your common, everyday actions slow you down, and then figure out how to perform those actions more efficiently. Fortunately, decades of software engineers have preceded us; chances are, others have already built the tools we need to accelerate our most common workflows. Often, all we need to do is to invest our time to learn them well.

Here are some ways you can get started on mastering your programming fundamentals:

- **Get proficient with your favorite text editor or IDE.** There are countless debates over which is the best text editor: Emacs, Vim, TextMate, Sublime,

or something else. What's most important for you is mastering the tool that you use for the most purposes. Run a Google search on productivity tips for your programming environment. Ask your more effective friends and co-workers if they'd mind you watching them for a bit while they're coding. Figure out the workflows for efficient file navigation, search and replace, auto-completion, and other common tasks for manipulating text and files. Learn and practice them.

- **Learn at least one productive, high-level programming language.** Scripting languages work wonders in comparison to compiled languages when you need to get something done quickly. Empirically, languages like C, C++, and Java tend to be 2–3x more verbose in terms of lines of code than higher-level languages like Python and Ruby; moreover, the higher-level languages come with more powerful, built-in primitives, including list comprehensions, functional arguments, and destructuring assignment. [22] Once you factor in the additional time needed to recover from mistakes or bugs, the absolute time differences start to compound. Each minute spent writing boilerplate code for a less productive language is a minute not spent tackling the meatier aspects of a problem.
- **Get familiar with UNIX (or Windows) shell commands.** Being able to manipulate and process data with basic UNIX tools instead of writing a Python or Java program can reduce the time to complete a task from minutes down to seconds. Learn basic commands like `grep`, `sort`, `uniq`, `wc`, `awk`, `sed`, `xargs`, and `find`, all of which can be piped together to execute arbitrarily powerful transformations. Read through helpful documentation in the `man` pages for a command if you're not sure what it does. Pick up or bookmark some useful one-liners. [23]
- **Prefer the keyboard over the mouse.** Seasoned programmers train themselves to navigate within files, launch applications, and even browse the web using their keyboards as much as possible, rather than a mouse or trackpad. This is because moving our hands back and forth from the keyboard to the mouse takes time, and, given how often we do it, provides a considerable opportunity for optimization. Many applications offer keyboard shortcuts

for common tasks, and most text editors and IDEs provide ways to bind custom key sequences to special actions for this purpose.

- **Automate your manual workflows.** Developing the skills to automate takes time, whether they be using shell scripts, browser extensions, or something else. But the cost of mastering these skills gets smaller the more often you do it and the better you get at it. As a rule of thumb, once I've manually performed a task three or more times, I start thinking about whether it would be worthwhile to automate it. For example, anyone working on web development has gone through the flow of editing the HTML or CSS for a web page, switching to the web browser, and reloading the page to see the changes. Wouldn't it be much more efficient to set up a tool that automatically re-renders the web page in real-time when you save your changes? [24] [25]
- **Test out ideas on an interactive interpreter.** In many traditional languages like C, C++, and Java, testing the behavior of even a small expression requires you to compile a program and run it. Languages like Python, Ruby, and JavaScript, however, have interpreters available allowing you to evaluate and test out expressions. Using them to build confidence that your program behaves as expected will provide a significant boost in iteration speed.
- **Make it fast and easy to run just the unit tests associated with your current changes.** Use testing tools that run only the subset of tests affected by your code. Even better, integrate the tool with your text editor or IDE so that you can invoke them with a few keystrokes. In general, the faster that you can run your tests, both in terms of how long it takes to invoke the tests and how long they take to run, the more you'll use tests as a normal part of your development—and the more time you'll save.

Given how much time we spend within our programming environments, mastering the basic tools that we use multiple times per day is a high-leverage investment. It lets us shift our limited time from the mechanics of programming to more important problems.

Don't Ignore Your Non-Engineering Bottlenecks

The best strategy for optimizing your iteration speed is the same as for optimizing the performance of any system: identify the biggest bottlenecks, and figure out how to eliminate them. What makes this difficult is that while tools, debugging workflows, and programming environments might be the areas most directly under your control as an engineer, sometimes they're not the only bottlenecks slowing you down.

Non-engineering constraints may also hinder your iteration speed. Customer support might be slow at collecting the details for a bug report. The company might have service-level agreements that guarantee their customers certain levels of uptime, and those constraints might limit how frequently you can release new software. Or your organization might have particular processes that you need to follow. Effective engineers identify and tackle the biggest bottlenecks, even if those bottlenecks don't involve writing code or fall within their comfort zone. They proactively try to fix processes inside their sphere of influence, and they do their best to work around areas outside of their control.

One common type of bottleneck is dependency on other people. A product manager might be slow at gathering the customer requirements that you need; a designer might not be providing the Photoshop mocks for a key workflow; another engineering team might not have delivered a promised feature, thus blocking your own development. While it's possible that laziness or incompetence is at play, oftentimes the cause is a misalignment of priorities rather than negative intention. Your frontend team might be slated to deliver a user-facing feature this quarter that depends on a piece of critical functionality from a backend team; but the backend team might have put it at the bottom of its priority list, under a slew of other projects dealing with scaling and reliability. This misalignment of priorities makes it difficult for you to succeed. The sooner you acknowledge that you need to personally address this bottleneck, the more likely you'll be able to either adapt your goals or establish consensus on the functionality's priority.

Communication is critical for making progress on people-related bottlenecks. Ask for updates and commitments from team members at meetings

or daily stand-ups. Periodically check in with that product manager to make sure what you need hasn't gotten dropped. Follow up with written communication (email or meeting notes) on key action items and dates that were decided in-person. Projects fail from under-communicating, not over-communicating. Even if resource constraints preclude the dependency that you want from being delivered any sooner, clarifying priorities and expectations enables you to plan ahead and work through alternatives. You might decide, for example, to handle the project dependency yourself; even though it will take additional time to learn how to do it, it will enable you to launch your feature sooner. This is a hard decision to make unless you've communicated regularly with the other party.

Another common type of bottleneck is obtaining approval from a key decision maker, typically an executive at the company. While I was at Google, for example, any user interface (UI) change to search results needed to be approved in a weekly UI review meeting with then-VP Marissa Mayer. There was a limited supply of review slots in those weekly meetings, coupled with high demand, and sometimes a change required multiple reviews.

Given that bottlenecks like these generally fall outside of an engineer's control, oftentimes the best that we can do is to work around them. Focus on securing buy-in as soon as possible. Mayer held occasional office hours, [26] and the teams who got things done were the ones who took advantage of those informal meetings to solicit early and frequent feedback. Don't wait until after you've invested massive amounts of engineering time to seek final project approval. Instead, prioritize building prototypes, collecting early data, conducting user studies, or whatever else is necessary to get preliminary project approval. Explicitly ask the decision makers what they care about the most, so that you can make sure to get those details right. If meeting with the decision makers isn't possible, talk with the product managers, designers, or other leaders who have worked closely with them and who might be able to provide insight into their thought processes. I've heard countless stories of engineers ready to launch their work, only to get last-minute feedback from key decision-makers that they needed to make significant changes—changes which would undo

weeks of engineering effort. Don't let that be you. Don't defer approvals until the end. We'll revisit this theme of early feedback in more depth in Chapter 6.

A third type of bottleneck is the review processes that accompany any project launch, whether they be verification by the quality assurance team, a scalability or reliability review by the performance team, or an audit by the security team. It's easy to get so focused on getting a feature to work that you defer these reviews to the last minute—only to realize that the team that needs to sign off on your work hadn't been aware of your launch plans and won't be available until two weeks from now. Plan ahead. Expend slightly more effort in coordination; it could make a significant dent in your iteration speed. Get the ball rolling on the requirements in your launch checklist, and don't wait until the last minute to schedule necessary reviews. Again, communication is key to ensure that review processes don't become bottlenecks.

At larger companies, fixing the bottlenecks might be out of your circle of influence, and the best you can do is work around them. At smaller startups, you often can directly address the bottlenecks themselves. When I started working on the user growth team at Quora, for example, we had to get design approval for most of our live traffic experiments. Approval meetings were a bottleneck. But over time, we eliminated that bottleneck by building up mutual trust; the founders knew that our team would use our best judgment and solicit feedback on the experiments that might be controversial. Not having to explicitly secure approval for every single experiment meant that our team could iterate at a much faster pace and try out many more ideas.

Given the different forms that your bottlenecks can take, Donald Knuth's oft-cited mantra, "premature optimization is the root of all evil," is a good heuristic to use. Building continuous deployment for search interface changes at Google, for example, wouldn't have made much of an impact in iteration speed, given that the weekly UI review was a much bigger bottleneck. Time would have been better spent figuring out how to speed up the approval process.

Find out where the biggest bottlenecks in your iteration cycle are, whether they're in the engineering tools, cross-team dependencies, approvals from decision-makers, or organizational processes. Then, work to optimize them.

Key Takeaways

- **The faster you can iterate, the more you can learn.** Conversely, when you move too slowly trying to avoid mistakes, you lose opportunities.
- **Invest in tooling.** Faster compile times, faster deployment cycles, and faster turnaround times for development all provide time-saving benefits that compound the more you use them.
- **Optimize your debugging workflow.** Don't underestimate how much time gets spent validating that your code works. Invest enough time to shorten those workflows.
- **Master the fundamentals of your craft.** Get comfortable and efficient with the development environment that you use on a daily basis. This will pay off dividends throughout your career.
- **Take a holistic view of your iteration loop.** Don't ignore any organizational and team-related bottlenecks that may be within your circle of influence.

This Doesn't Have to Be the End

Hey there dear reader,

You might have reached the end of the sample chapter, but this doesn't have to be the end of our conversation. You can get the rest of the book and other special goodies at https://effectiveengineer.com/book.

It's also a bit rude of me to have done all the talking so far. I'd love to hear from you, whether it's feedback on what you liked and didn't like from the book, how this resource could be more useful, or just to say hi and introduce yourself.

And if there are any challenges you're facing at work or pain points in your engineering career where you're looking for some guidance, do let me know. Even if I can't help you directly, maybe I can point you to a useful resource.

You can leave your name, email, and any thoughts on this survey, or just send me a note at hi@effectiveengineer.com.

I read every email from my readers.

Cheers,
Edmond